

Automatic Transformation of UML to Decision Models and Assessment of Architectural Decisions

Sunny Wong
Drexel University
Philadelphia, PA, USA
sunny@cs.drexel.edu

Warren Baelen
Drexel University
Philadelphia, PA, USA
st95b589@drexel.edu

Yuanfang Cai
Drexel University
Philadelphia, PA, USA
yfcai@cs.drexel.edu

ABSTRACT

The *design structure matrix* (DSM) and its formal counterpart, the *augmented constraint network* (ACN), have been used to explicitly model software design decisions, design rules and the resulting modular structure. To address the difficulty of modeling software design in terms of decisions, we formally define the decisions and their assumption relations embedded within UML class diagrams and automatically transform a UML class diagram into an augmented constraint network, from which a DSM can be automatically derived. We also use a real-world software project to assess the difference between syntactical DSMs derived from source code with UML-derived DSMs in terms of their ability of assessing software architecture. The result shows that the UML-derived ACN and DSM models can more effectively help designers to isolate the impact of each individual architectural design rule and feature, assess the effectiveness of refactoring, and locate the maintenance activities that may cause modularity decay.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Maintenance and Enhancement—*refactoring*; D.2.10 [Software Engineering]: Design—*modularity assessment, design modeling*

Keywords

design rule theory, modularity assessment

1. INTRODUCTION

The *design structure matrix* (DSM) [3] and *augmented constraint network* (ACN) [7] have been useful in expressing and evaluating software modular structure [16, 19, 20], and explaining software evolution phenomena [15, 16]. Despite their promise, constructing these analytical decision models is not so simple and straightforward. Manually modeling software design with an ACN, from which a DSM can be automatically derived, requires designers to think in terms of

design decisions and constraints upon those decisions. And manually marking the dependencies in a DSM is error-prone and subject to ambiguity [7]. Prevailing DSM-based approaches use DSM models reverse engineered from source code, which we call a *syntactic DSM* (or SDSM for short). However, research has shown that important design decisions and relations may not be explicitly available in source code and thus could be missing from a SDSM. These difficulties limit the application of these models in practice. To address these problems, we present an approach for converting prevailing design models into these analytical decision models—ACN and DSM—and show that these automatically constructed analytical decision models offer additional insight, over SDSM models, into the modularity of a design.

The *unified modeling language* (UML) [17] is a well-understood and widely-used modeling technique. Although UML models are not designed to generally model design decisions, important design decisions and their relations are embodied within these models. Different from traditional dependency modeling, we view each class as having two kinds of decisions, its interface that are publicly visible and its hidden implementation. For example, a line in a UML class diagram showing that a class A calls methods in another class B can be viewed as showing that there exists a constraint between the decisions on the methods exposed through B's publicly-exposed interface and A's implementation decision. In other words, changing the decision regarding the method signatures on B could require changes to the implementation of A that calls those methods. Motivated by this observation, we contribute a formalization of UML class diagrams to ACN models, which explicitly expresses the semantics of UML elements as design decisions and their constraints.

Although such a formalization does not entirely capture all the decisions and constraints in a design, it captures more implicit and indirect dependencies than are not shown in prevailing reverse-engineered SDSMs. We show that these additional dependencies are useful for certain modularity assessments. We developed a tool, called *uml2acn*, that automatically derives an ACN model from a UML class diagram built from prevailing modeling tools, such as Rational Rose¹ and Dia². The ACN model then can serve as input to Minos [23], a tool for ACN analysis and DSM generation. Since DSMs can also be directly derived from UML class diagrams in the same way they can be derived from source code, we also explore if our formalization to ACN offers benefits over a direct conversion of source code or UML to DSM in as-

¹<http://www-01.ibm.com/software/rational/>

²<http://projects.gnome.org/dia/>

sessing software modularity. In particular, we ask, if using the UML-transformed ACN and DSM models:

- Are we able to better assess whether a refactoring activity is successful or not?
- Are we able to better assess the quality of architectural decisions?
- Are we able to identify problematic decisions easier?

We note that in most of the prior work that use SDSMs to assess software architecture and evolution, either the architecture changes were so significant or the refactoring was so successful that SDSMs can reveal the prominent structure differences; or the architecture changes accumulated over time and the structure differences become obvious. In reality, however, it is more often that two architecture structures may reveal similar modularity properties—a refactoring activity may not be very successful or a successful refactoring is offset by low quality feature addition. In these cases, it is more important to isolate and locate problematic features and maintenance activities than to measure overall modularity variations. In this paper, we show that in these scenarios, when traditional metrics (e.g., coupling, cohesion) do not significantly differ in SDSMs, the DSMs derived from ACNs formalized from UML diagrams (or ADSM for short) help us identify problematic design decisions more easily.

To evaluate the modularity assessment ability of ADSMs, we applied our approach to a real industrial project during a release cycle. During this time, not only were new features added to the system but, simultaneously, legacy code was refactored due to observed inefficiencies caused by modularity decay. As the product was released, the designers suspected that some of the new features were implemented in a “quick and dirty” manner and some of the refactorings were not performed properly. We compared the use of SDSMs and ADSMs in aiding the designers to detect problematic maintenance activities, answering these questions:

- Can ADSM modeling better reveal the quality of overall modularity improvement before and after the refactorings?
- Can ADSM modeling better isolate the features or components that cause modularity decay?
- Can ADSM modeling better assess whether a refactoring was successfully applied?
- Can ADSM modeling better assess the quality of the *design rules* [3], or interfaces, introduced for a refactoring? That is, are these design rules following major design principles and consistent with how the system changes?

By modeling the software design at both the beginning and end of the release cycle, using both ADSMs and SDSMs, we compare the differences between each pair of DSMs using various metrics to measure the design modularity. Our results show that while the ADSM and SDSM models do not show much difference in terms of answering the first question, but an ADSM outperforms an SDSM in identifying and isolating problematic design decisions, answering the other three questions.

The rest of the paper is organized as follows. Section 2 addresses related work. Section 3 describes our UML formalization and modularity assessment approach. Section 4 presents our evaluation and Section 5 discusses our threats to validity and future work. Section 6 concludes.

2. RELATED WORK

This section reviews related work, and differentiates our UML formalization and modularity assessment technique from existing approaches.

UML Formalization and Analysis.

Many researchers have proposed various approaches for formalizing UML (e.g., Evans et al. [11], Baresi and Pezzè [4], Anastasakis et al [1], Varró et al. [21]). The purpose of such formalizations is often to detect inconsistencies and errors in UML diagrams. In contrast, the purpose of our UML formalization is not to check for inconsistencies, but to enable software practitioners trained with UML modeling to leverage ACN-based modularity and evolution analyses, such as Baldwin and Clark’s net option value analysis [3], design change impact analysis [5,7], modularity conformance checking [13], and *design rule hierarchy* (DRH) analysis [22]. To the best of the authors’ knowledge, there are no existing formalizations of UML using the ACN model.

Various approaches have been proposed for documenting (e.g., Capilla et al. [9]) and reverse engineering (e.g., Jansen et al. [14]) architectural design decisions and assumptions. Since the ACN model we use in this paper models design decisions and their relations, our UML formalization can also be viewed as a reverse engineering of design decisions. Existing design decision recovery approaches often require participation of the original software designers to solicit design decisions. In contrast, although our UML formalization approach does not recover all design decisions, it automatically recovers important indirect and implicit decisions embedded in UML models that are useful for modularity assessment. Additionally, prevailing design decision recovery approaches use informal models for representing the captured design decisions and their relations. By transforming a UML class diagram into an ACN model, our approach enables formal and thus automated modularity analyses.

Software Design Metrics.

Numerous metrics on software stability and modularity have been proposed (e.g., Chidamber and Kemerer [10]) and often are applied at the source code level. Our recent work also proposed a suite of metrics [6, 20] based on Baldwin and Clark’s design rule theory and the *design rule hierarchy* clustering [22], such as the *Independence Level* (IL) metric that measures how well an architecture can support independent module substitution, and thus option value generation. In this paper, we show that after transforming a UML class diagram into an ACN and DSM, these metrics can be leveraged to offer more insights on the modularity structure without manually constructing decision models, and in particular, to separate the impact of each individual features or design rules. Different from our work of assessing design modularity and stability based on UML component diagram transformed ACN models [20], the assessment approach presented in this paper does not required the mapping of design components with the high-level *features* that they are responsible for.

Bahsoon and Emmerich [2] proposed a framework for assessing the stability of an architecture using *real option theory*. By formalizing UML class diagrams into ACN models, our approach allows a similar *net option value* analysis [3] to be performed on a design. Both *real option* and *net option*

analyses require the difficult estimation of certain parameters, such as *cost* and *technical potential*. Although using our design rule hierarchy to assess modularity may not be as accurate as these existing approaches, our approach does not require the estimation of such parameters.

DSM Analysis.

Various products (e.g., Lattix LDM³, NDepend⁴) are available for reverse engineering a DSM from source code or compiled binaries. These tools usually support traditional metrics calculation, such as coupling, cohesion and stability. Different from their definitions of elements and dependencies, our approach uniquely separates each class into two DSM variables, one for the publicly-exposed interface and one for the private implementation details. From the UML-transformed ACN model, the underlying constraint-based reasoning automatically reveals many more *indirect* or *implicit* dependencies that are not shown in reverse-engineered DSMs. In this paper, we show that the explicit revelation of *indirect* or *implicit* dependencies and the separation of interface and implementation, can provide additional useful insights for software architects and maintainers. We also compare our UML-transformed ACN and DSM models with the reverse-engineered DSMs in term of their architecture analysis abilities.

3. APPROACH

In this section, we present an overview of our UML formalization and *design rule hierarchy* (DRH) based [23] modularity assessment approach.

3.1 Framework Overview

Our modularity assessment approach takes a model of the software structure, such as a UML class diagram, as input and outputs a *design structure matrix* (DSM) model that is clustered based on an automatically derived *design rule hierarchy* [23]. Figure 1 shows an overview of our modularity assessment approach. Since many projects may not have UML class diagrams available, we contribute two tools, *Moka* and *Grenada*, for reverse engineering UML class diagrams from compiled Java and .NET binaries. The *uml2acn* tool formalizes a UML class diagram into an *augmented constraint network* (ACN) model [7, 8], a design model that formalizes the key concepts of Baldwin and Clark’s design rule theory [3]. *Minos* [23] is our core tool for performing ACN analysis and automatically deriving a *design structure matrix* (DSM) model from an ACN.

Given a set of compiled binaries or a UML class diagram, we first convert it to an intermediate format, which we call the *DEPS* file format. This format contains the information of a class diagram, in a simple, standard format so that the *uml2acn* tool can uniformly formalize the different input class diagram formats into an ACN model. This intermediate format allows *uml2acn* to accept multiple input formats (e.g., XMI [18], Rational Rose).

Given an ACN model, the *Minos* tool can automatically compute a design rule hierarchy and derive a DSM model [8, 22, 23] to show the structure of dependencies. By deriving a DSM from an ACN, rather than directly from the class diagram or source code, our approach explicitly exposes *indirect*

and *implicit* dependencies between components, which can be important in performing maintenance tasks. We cluster the DSM using the DR hierarchy in order to show areas of the design that are potentially problematic for independent and parallel maintenance.

3.2 Background

This subsection describes the concepts of design structure matrix, augmented constraint network and design rule hierarchy, used in our UML formalization and modularity assessment technique. We use the simple UML class diagram shown in Figure 2 as an illustrative example.

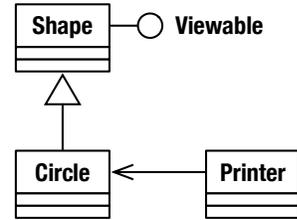


Figure 2: Example UML Diagram

Design Structure Matrix.

A *design structure matrix* (DSM) is a square matrix in which rows and columns are labeled with design dimensions in which decisions have to be made. We refer to the row/column labels as *variables* of the DSM. In contrast to prevailing DSM modeling, we view each class as having *two* design dimensions, an interface (ending with *_interface*) and an implementation (ending with *_impl*). Figure 3 shows a DSM automatically derived, using our UML formalization technique, from the UML class diagram in Figure 2. A marked cell in a DSM models that the decision on the row depends on the decision on the column. For example, the cell in row 7, column 2 indicates that the implementation of the class *Printer*, *Printer_impl*, depends on the interface of the class *Shape*, *Shape_interface*.

	1	2	3	4	5	6	7
1 Viewable_interface	.						
2 Shape_interface	x	.					
3 Circle_interface	x	x	.				
4 Shape_impl	x	x		.			
5 Circle_impl	x	x	x	x	.		
6 Printer_interface						.	
7 Printer_impl	x	x	x			x	.

Figure 3: Example Design Structure Matrix

Building on DSM models, Baldwin and Clark proposed the notion of design rules as stable design decisions that decouple otherwise coupled design decisions, hiding the details of other components. Examples of design rules in software include abstract interfaces, application programming interfaces (APIs), a data format agreed among development teams, and even naming conventions. Broadly speaking, all the non-private part of a class that are used by other classes can be seen as design rules.

DSM modeling can capture the concept of modules and design rules, as well as their decoupling effects. Modules are

³<http://www.lattix.com/products/ldm-ldv>

⁴<http://www.ndepend.com/>

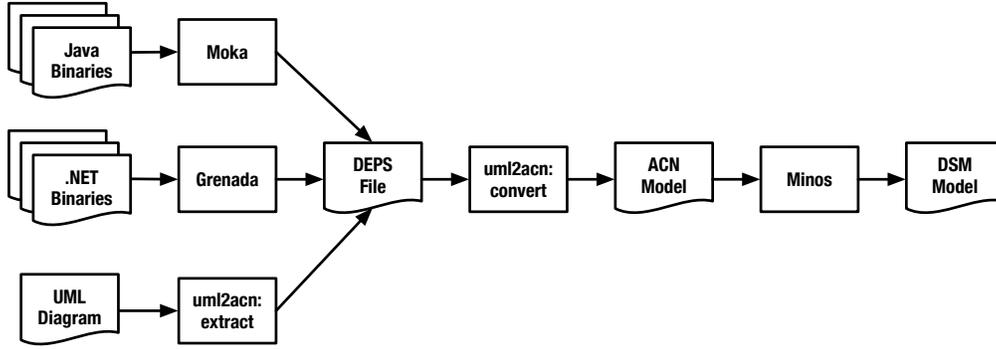


Figure 1: Approach Overview

represented as blocks along the diagonal and design rules are captured by asymmetric dependencies that decouple modules. However, manually identifying design rules and their impact scopes can be tedious and error-prone. To address this issue, we recently developed an algorithm to automatically compute the *design rule hierarchy* (DRH) of a design, and thereby identify the design rules [23].

Design Rule Hierarchy and Associated Analyses.

The DSM shown in Figure 3 is clustered into a two-layer DR hierarchy [23]. Each *layer* is shown as a shaded block of cells. For example, the last four variables form the second layer of the hierarchy. This layer consists of two *modules*, as shown by the two boxes within the shaded region, each with two variables. The top layer of the DSM consists of a single module, with three variables that are the design rules for the rest of the system. Each module clusters a set of design decisions that should be made together. Modules within a layer have no dependencies upon each other and have the potential to be assigned as independent, parallel tasks. The DRH in Figure 3 shows that after the DRs are determined, the **Printer** class can be implemented independent of the design and implementation of the shapes.

In a DRH-clustered DSM, modules only depend on modules in layers to their left in the DSM, and the layered structure hints at a development order which maximizes task parallelism. The last layer of a DRH consists of independent modules that, as long as the decisions on previous layers are respected, each provides an option to be substituted with better versions without influencing the rest of the system, hence generating option values [3]. The more dimensions in a system that fall into the last layer, the higher option value can be generated. Our recent work proposed the *Independence Level* (IL) metric to measure the percentage of the last layer variables. The number of layers within a DRH also reveals important modularity property. If a DRH has many layers then the design potentially supports less task parallelism and is less likely to be well-modularized.

We see from Figure 3 that although the public interface of any class can potentially be a design rule, they are not necessary design rules. For example, the **Printer** class’s interface is not a design rule because it does not influence any other parts of the system, other than its own implementation. The DRH is calculated based on how decisions make assumptions upon each other as modeled in the augmented

constraint network model.

Augmented Constraint Network.

Our recent work has shown that manually-constructed DSMs can be ambiguous and error-prone [5, 7]. To address this problem, we have developed a design representation called the *augmented constraint network* (ACN) to precisely represent the relations among design decisions using logical constraints. Figure 4 shows part of the ACN model automatically derived from the UML class diagram shown in Figure 2. An ACN consists of a *constraint network* that models design decisions and their relations, a *dominance relation* that formalizes design rules, and a *cluster set* in which each cluster represents a different way to partition a design. A constraint network consists of a set of design *variables*, which model design dimensions, environment conditions, and their domains; and a set of logical *constraints*, which model the relations among variables. The main relations we model using an ACN is the *assumption* relations among design variables. Figure 4 is a partial ACN model derived from the UML class diagram shown in Figure 2.

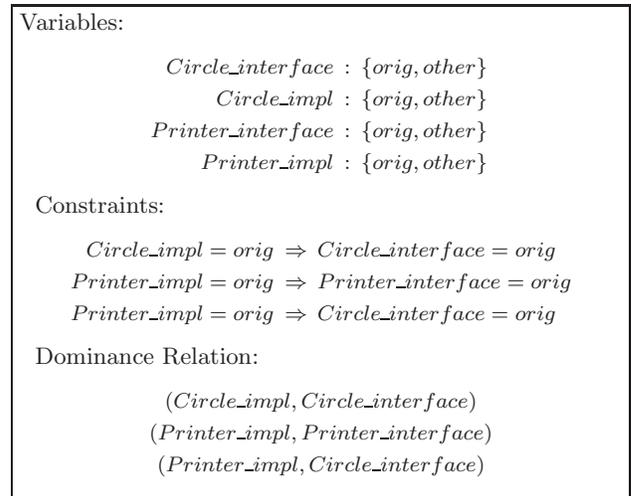


Figure 4: Example Augmented Constraint Network

Besides the ACN-based analysis techniques presented in our previous work [5, 6, 8, 20], we observe that UML-transformed ACN models allow us to assess the structural impact of each feature separately, by removing that the variables

modeling the feature (either from ACN or from UML) and recalculate the remaining DSM and the associated modularity properties. If the remaining modular structure improves after removing the feature, it indicates that the feature was causing modularity degradation. We illustrate this approach in detail in our real-world case study.

3.3 UML Formalization

This subsection describes our UML-to-ACN formalization technique. The basic idea for converting UML to ACN is to formalize each relation (e.g., generalization and aggregation) using a constraint network and dominance relation. Due to space considerations, we only present the formalizations for several UML relations. Our technique currently supports formalization of classes, interfaces, and the major binary relations of class diagrams: generalization, realization, dependency, aggregation, composition, navigable association, and owned element. Details of the formalization for all these relations are found elsewhere [12].

From the design decision perspective, each UML class consists of two design dimensions: the interface dimension, and the implementation dimension. As a result, we model a class, **A**, using two variables: $A_interface$ and A_impl . Each dimension can vary, so we model the domain of each variable as having at least two values (*orig*, *other*), where *orig* models the current decision and *other* models an unelaborated new choice for the decision.

The ACN translated from class **A** is shown as below. Besides the two variables, we use a logical expression to model that the implementation of **A** makes assumptions about its interface. We also assume that the interface of a class dominates its implementation, which is translated into a dominance relation. As a result, we translate the class **A** into the following ACN:

Variables:
 $A_interface : (orig, other)$
 $A_impl : (orig, other)$

Constraints:
 $A_impl = orig \Rightarrow A_interface$

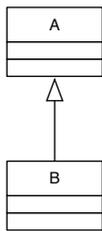
Dominance:
 $(A_impl, A_interface)$

Generalization.

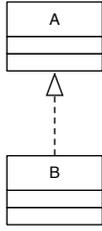
Table 1 shows a *generalization* relation depicted in UML, in which **A** is the general element and **B** is the specific element. Since **B** inherits from **A**, the decision on **A**'s interface dominates and influences both **B**'s interface and implementation, as modeled using the first two logical constraints and first two dominance pairs shown in the table. The decision on A_impl also influences B_impl as expressed in the third constraint. A_impl also dominates B_impl because a change in **B**'s implementation should not force changes in **A**'s implementation, which may be inherited by other classes. The dominance relation is constructed accordingly.

Realization.

Table 2 shows a *realization* relation depicted in UML, in which **A** is the interface and **B** is the implementing class. Since **B** realizes **A**, the decision on **A**'s interface dominates and influences both **B**'s interface and implementation, as modeled using the two logical constraints and two dominance pairs

UML	Augmented Constraint Network
	Constraint Network: $B_impl = orig \Rightarrow A_interface = orig$ $B_interface = orig \Rightarrow A_interface = orig$ $B_impl = orig \Rightarrow A_impl = orig$ Dominance Relation: $(B_impl = orig, A_interface)$ $(B_interface, A_interface)$ $(B_impl = orig, A_impl)$

shown in the table.

UML	Augmented Constraints Network
	Constraint Network: $B_impl = orig \Rightarrow A_interface = orig$ $B_interface = orig \Rightarrow A_interface = orig$ Dominance Relation: $(B_impl = orig, A_interface)$ $(B_interface = orig, A_interface)$

4. EVALUATION

The purpose of our evaluation is to assess whether the formalization of UML class diagrams to ACN models can help designers better evaluate maintenance activities and architectural decisions, compared to SDSMs reverse-engineered from the source code. Concretely, we explore the following questions:

Q1: When compared to SDSMs and traditional modularity metrics, do ADSMs better reveal overall modularity variations before and after refactoring? We use both SDSMs and ADSMs to represent the design before and after refactoring, and use associated metrics to see if they can tell how the modularity properties vary.

Q2: When compared to SDSMs, do ADSMs more effectively isolate the features that may lead to modularity decay? Since features are often implemented by developers with varying levels of experience and capability, it is possible that some features are added in a “quick and dirty” manner. We want to see whether and how the DSM models can help designers identify problematic features.

Q3: When compared to SDSMs, do ADSMs provide a better assessment of whether each refactoring design rule was successfully applied? When refactoring a design, maintainers may often introduce a design rule to decouple components; we call such a design rule as a *refactoring design rule*. It is possible that not all design rules are faithfully implemented and that obsolete design rules are no longer followed. That is, a design rule may be introduced during a refactoring to remove the coupling between several components. However, designers need to be sure that, after the refactoring, these components are no longer dependent on each other. It is important to assess the implementa-

tion of each individual design rule in order to reduce future maintenance efforts.

Q4: When compared to SDSMs, do ADSM help designers more easily assess the quality of the refactoring design rules themselves? When a design rule is determined, the designer makes assumptions about which part of the system should change and which part should remain stable. If the assumptions are incorrect, then the design rule itself may not be stable. In addition, the design rule itself may violate major design principles. Identifying these problematic design rules is important for future development activities.

4.1 Subjects

We studied a real-world product to answer these questions. The project in question has gone through four years of evolution starting from a prototype stage. In its latest release, unit testing became a requirement mandated by a company-wide policy. The development team faced difficulty in generating unit test cases for their legacy code because of the highly coupled architecture. At the same time, they also needed to deliver several new features to market. The project lead thus decided to refactor the legacy code by applying several new design rules, including introducing design patterns and interfaces to better modularize the system. The key data structure and pattern-related code were first implemented. After that, the developers of the new features were asked to follow these new design rules. At the same time, the remaining legacy code was also refactored to follow the new design rules. Feature additions and legacy code refactoring were assigned to eight different developers and were conducted simultaneously.

Eight months later, the new project needed to be delivered. However, the team still had difficulty conducting unit tests efficiently because of coupling issues. That is, despite the refactorings, no significant modularity enhancement was observed. The designers also tried to use prevailing modularity metrics, such as coupling and cohesion, to measure the architecture variation but the differences were still unclear.

Several reasons were suspected: (1) A problem of the new architecture: Are the new architecture design rules not good enough to improve system modularity? However, the newly applied design patterns were well-know solutions to the problems shown in the legacy code, and the coupling should have been significantly decreased where the design patterns were applied. It was hard to believe that the application of design patterns did not provide any modularity improvement. (2) A problem of the developers: Since the capability of developers vary significantly, it is possible that some developers made some quick and dirty decisions that offset the modularity improvement brought by the new architecture. (3) The problems may come from both aspects. In this case, it is more important for us to identify and locate *where* the problems are than to merely measure the overall modularity variation.

For proprietary issues, we are not allowed to reveal the name of the product. We simply call the system before refactoring as *pre-TDD* (TDD stands for Test Driven Development), and the new release as *post-TDD*. For the rest of the section, we assign each feature and design rule an ID instead of using their original name. The product is written on the .NET framework. The pre-TDD version had 311 classes and 29 KLOC. The post-TDD version had 728 classes

and 36 KLOC.

4.2 Evaluation Procedure

To understand whether the overall modularity improved or degraded from pre-TDD to post-TDD (Q1), we used NDepend to generate SDSMs from compiled assemblies, used Grenada to reverse engineer the source code into UML class diagrams, use uml2acn to generate ACN models, and use Minos to derive ADSMs. We generated both SDSMs and ADSMs for both pre-TDD and post-TDD to see whether the associated metrics can tell us how the modularity structured changed.

For SDSM, we consider the following standard metrics that can be calculated by NDepend:

Afferent coupling (Ca): The number of types outside a component that depend on types within this component. High afferent coupling indicates that the concerned components have many responsibilities.

Efferent coupling (Ce): The number of types inside a component that depend on types outside this component. High efferent coupling indicates that the concerned component is dependent on many other components.

Cohesion (H): The average number of internal relationships per type. Let R be the number of type relationships that are internal to this component. Let N be the number of types within the component. Then the cohesion is computed as $H = (R + 1) / N$. The relational cohesion represents the relationship that a component has to all its types. As classes inside a component should be strongly related, the cohesion should be high. On the other hand, too high values may indicate over-coupling. A recommend range for H is 1.5 to 4.0.

Instability (I): The ratio of efferent coupling Ce to total coupling (both efferent and afferent): $I = Ce / (Ce + Ca)$. This metric is an indicator of the component's resilience to change. The range for this metric is 0 to 1, with $I = 0$ indicating a completely stable component and $I = 1$ indicating a completely unstable component.

For ADSM, we consider the following metrics used in our prior work [20]:

Number of layers (#Level): The number of layers, or levels, in the design rule hierarchy. Since the modules within a layer of the DRH can be assigned as parallel tasks, the larger the number of layers there are, the less potential parallelism for task assignments there is.

Independence level (IL): The ratio of design decisions in the last layer of the DRH to the total number of design decisions in the ACN. Since nothing depends on the modules in the last layer of the DRH, these modules can be independently substituted for better implementations. Hence, the higher the independence level is, the more of a system can evolve independently without impacting other parts, and the better the system is modularized.

To assess the ability of a DSM model to isolate features (Q2), our strategy is to see if it is possible to remove each feature from both DSM models. If possible, we determine how the modularity metrics will then change. If the changed metric values shows significant modularity improvement after removing the feature, it indicates that the feature was added in a sub-standard way. We then ask the designer of the project to confirm the observation.

To assess whether each design rule is successfully applied (Q3), we first worked with the designer of the project to

Table 4: SDSM Metrics

	<i>PreTDD</i>	<i>PostTDD</i>
# Afferent Coupling	311	728
# Efferent Coupling Pair	400.5	272.6
# Cohesion (1.5-4.0)	3.465	2.852
# Instability (0-1)	0.445	0.84

Table 5: DRH Metrics

	<i>PreTDD</i>	<i>PostTDD</i>
# Classes	311	728
# Levels Pair	5	12
# Last Level Classes	185	398
# IL	0.595	0.547

identify all the refactoring design rules. Then we query the DSM models to see if each design rule is serving its decoupling purposes. That is, if the two components that are supposed to be decoupled became decoupled after introduction of the design rule, or if unexpected dependencies were introduced. In Table 3, the first columns are the IDs of all the new design rules in post-TDD. The second column explains their respective purposes. The third column explains how these design rules were reflected in code.

According to the purposes and how they are reflected in the code, we separate these design rules into single-purpose design rules and multiple-purpose design rules. DR1, DR2, DR3, DR4, DR5, DR9 and DR10 are designed for only one purpose. DR6, DR7 and DR 8 are designed for two main purposes: (1) to decouple client and internal data and (2) to separate the responsibilities of a complex object into multiple interfaces or classes. We query both DSMs to see if the classes that are supposed to be separated were indeed separated. For DR10, we query the DSM to see if the classes that are supposed to be implemented the interface, and no other classes, really depend on this interface.

To assess the quality of design rules themselves (Q4). Similar to our approach of assessing the quality of features, we first try to remove each design rule from both DSM models to see how the modularity metrics will change. We also observe whether and which DSM models can better reveal the quality of these design rules against major design principles, such as the dependency inversion principle and information hiding principle.

4.3 Results

This section presents the results of our evaluation by answering the evaluation questions proposed at the beginning of the section.

Overall modularity properties (Q1).

Table 4 and Table 5 list the metric values for the SDSM and ADSM respectively. Three out of the four SDSM metrics indicate modularity degradation in postTDD: afferent coupling increased, less cohesion and less stable. Only efferent coupling decreased. The ADSM metrics consistently show that the modularity degraded after the release cycle: the number of layers more than doubled and the independence level decreased from 0.595 to 0.547. The result shows that both metric suites indicate modularity degradation, and these measurements complement with each other.

Isolation of Features (Q2).

We assess the impact of each individual feature by observing how modular properties change if the feature is removed.

From our experiments, for both DSM models, removing the columns and rows directly do not work. For the ADSM, removing columns and rows would incorrectly leave implicit and indirect dependencies. For SDSM, since the tool we use only calculates the metrics from source code assemblies, we could not get correct measurements after removing the columns and rows.

For SDSM, the only viable way is to remove the feature from the code, recompile and regenerate the DSM. We experimented on one feature; it took a long time to trim the code and make the rest of the code compilable due to problems such as reconfiguring the program, and adding extra code to make the rest compilable. The designer of the project decided that code trimming and recompiling was too expensive, and the resulting trim code may not correctly reflect the structure impact of the removed feature. The conclusion is that, removing features from SDSM appears to be infeasible in practice.

It turned out that the only feasible way was to remove the corresponding ACN variables or from reverse-engineered UML model. It only took a few minutes to remove each feature and regenerate the ADSM. It is not a problem that the remaining parts, once mapped to the code, are not compilable because we just need to assess their structural impact. The result shows that the number of layers do not change after removing any of the three features, but the IL changes as shown in the first three data points in Figure 5.

In Figure 5, a positive value means that after removing the feature or design rule, the modularity of the system is improved, showing that it is contributing to modularity degradation. A negative value means that adding the feature or design rule improves the system modularity. The larger the absolute value, the more significant the contribution is.

The figure shows that after removing the first feature (32 variables were removed), the IL improved significantly, which means that compared to the other two features, the first feature was most problematic and contributed most to the observed modularity degradation. This observation was confirmed by the designer. According to the designer, feature 1 was hastily implemented in the end of the product's prior release cycle. It was an area of code that was avoided by the developers because it was difficult to understand due to both the number of classes involved and a tall inheritance chain. Basically, the feature acted as a dialog to allow users to create an expression to reduce the number of records returned in a report. It had many dependencies with the user interface class, the query model and the expression model classes. The designer was interested to see the ADSM confirm something that had always been suspected.

Effects of Design Rules (Q3).

The last two columns of Table 3 show that, from SDSMs, all these design rules were successfully applied, and achieved their purposes. From the ADSM assessment, two single-purpose design rules, DR5 and DR10, did not achieve their purposes. None of the two-purpose interfaces achieved their purpose for separation of concern successfully.

The difference comes from the indirect and implicit dependencies picked up by the ADSMs. As an example, Figure 6 shows a partial ADSM in which the cells with gray background are the implicit or indirect dependencies picked up by our UML-transformed ACN, but are not shown in the corresponding reverse engineered SDSM.

Table 3: Refactoring Design Rules

ID	Purpose	In Code	SDSM		ADSM	
			P1	P2	P1	P2
DR1	Limit call to 3rd party components	2 interfaces Added	yes	n/a	yes	n/a
DR2	Remove extensive use of singleton	2 classes Added	yes	n/a	yes	n/a
DR3	Decouple metadata and their client	2 interfaces added. Prior references to 7 objects are replaced with references to the 2 new interfaces	yes	n/a	yes	n/a
DR4	Decouple data sources and their representations	A set of interfaces were added to apply visitor pattern	yes	n/a	yes	n/a
DR5	Decouple a complex object into multiple interfaces so that each interface take care of single responsibility	references to two classes are replace with references to multiple new interfaces	yes	n/a	no	n/a
DR6 (2)	separate models and their representations	model-view-presenter pattern is used: 8 new interfaces were introduced to replace two complex objects	yes	yes	yes	no
DR7 (2)	heavy-loaded classes are separated	a new interface is added and the relation of a set of classes are changed	yes	yes	yes	no
DR8 (2)	decouple data source from UI component	a set of new interfaces and classes are added	yes	yes	yes	no
DR9	decouple data from report format	3 classes added	yes	n/a	yes	n/a
DR10	to make all the query model have the same interface	1 interface added and a number of existing classes changed	yes	n/a	no	n/a

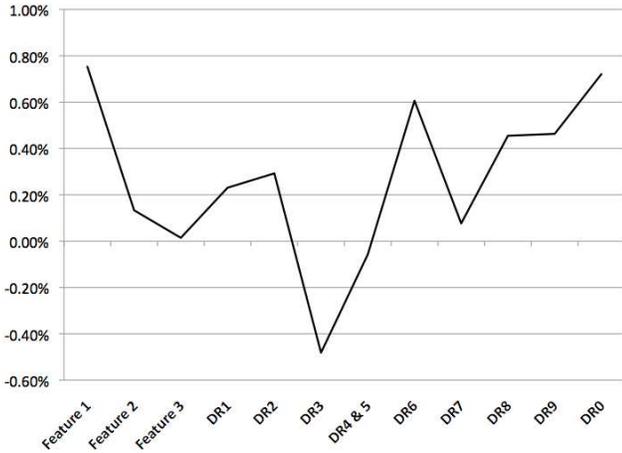


Figure 5: Increase in IL from Feature Removal

Two components that were suppose to be decoupled by the design rules were actually still coupled implicitly or indirectly. By examining these extra dependencies, the designers were able to identify many surprising dependencies that makes unit testing difficult. For example, after applying DR8, *XResultSetImpl* is unexpectedly shown to be dependent on *CDLCell*, which is not shown in the syntactic view, but these two classes cannot be tested independently. The reason is that *XResultSetImpl* inherits the implementation of its parent, an abstract class, which depend on *CDLCell* erroneously. This problem is caused by a deep inheritance hierarchy, and the violation of dependency inversion principle. By examining these extra dependencies, the designer was able to explain why testing was still difficult after refactoring and to pinpoint the location of the problem.

Quality of the Design Rules (Q4).

Given the difficulty of removing design rules from SDSM, we could only answer Q4 using ADSMs. Table 6 shows the ADSM metric values after removing each design rule from post-TDD. Retrospectively, the DR10 (adding a unified interface) was a poor decision because it violated the Interface Segregation principle. In our experiments, after removing DR10, the number of layers dropped from 12 to 10, and the IL increased, which meant that this design rule significantly constraints the design space and decreases modularity. The development team has realized the problem and removed

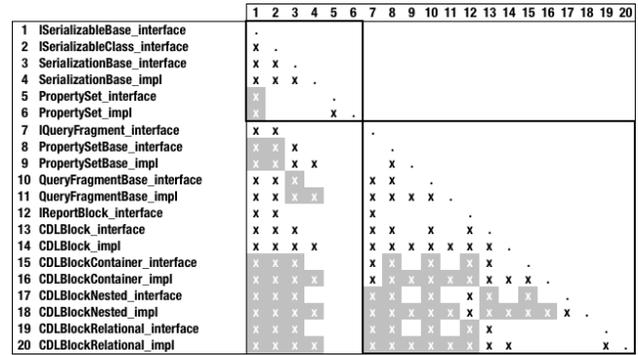


Figure 6: Increase in IL from Feature Removal

this design rule in the code while this study was conducted. As a result, we use the metrics of *postTDD-DR10*, shown in the first column of Table 6, as the baseline to compare modularity variations.

Given the convenience of removing design rules, the designer removed an old design rule (DR0) to see its impact, because this design rules was creating many testing issues and drew the designer’s attention. Figure 5 shows how IL changes after removing each design rules.

The figure shows that the old design rule DR0 was indeed most problematic because after removing it, the number of levels reduced to 9 and the IL significantly increased, which confirmed the fact that this DR was actually causing problems. The most successful design rules are DR3 and DR4&5 because after removing them, the modularity was degraded, especially for DR3. Since both DR4 and DR5 worked on the same class, we combined them. The designers confirmed that applying these design rules made their work easier, which was especially true for DR3.

A more important step was to locate why adding some of the design rules were decreasing the modularity. After examining the ADSMs clustered according to the design rule hierarchy, we observe the following symptoms of poor design:

1. The appearance of *_impl* in higher layers of the DRH. This means that an implementation is inherited and supposed to be stable, which violates the dependency inversion principle and information hiding principle. Table 7 and 8 show the number of *_impl* variable appeared in each layer for both pre-TDD and post-TDD. The tables show that in post-TDD, many implementation decisions became design rules. After examining the correspond DRH-clustered

Table 6: Design Rule Implementation Assessment

PostTDD-DR10		DR1	DR2	DR3	DR4 & 5	DR6	DR7	DR8	DR9	DR0
721	# of Classes	719	719	678	593	706	720	663	715	667
10	# of Layers	10	10	10	9	10	10	10	10	9
398	# of last layer classes	398	399	371	327	394	398	369	398	373
0.5520	IL	0.5543	0.5549	0.5472	0.5514	0.5581	0.5528	0.5566	0.5566	0.5592
0.53%	Δ IL	0.23%	0.29%	-0.48%	-0.06%	0.61%	0.08%	0.45%	0.46%	0.72%
	Δ size	2	2	43	128	15	1	58	6	54
	Δ last layer classes	0	1	-27	-71	-4	0	-29	0	-25

ADSMs, the designer of the project identified two reasons (1) in the legacy code, a large number of abstract classes with implementations were inherited. (2) when new features were added, developers tended to inherit from an existing class for convenience, and thus implicitly depended on many implementations unnecessarily. From the DRH, the designers were able to pinpoint where the violation happened by checking which class’s implementation was in a high layer of the DRH. For example, DR9 was realized by three new classes, and one of their implementations appears in the second layer of DRH. After examining the corresponding code, the designer realized that the class’s implementation, which was not stable, was inherited. In DR0, the implementation of a heavily used `Expression` class was in layer 4. This class, however, turned out to be very unstable, which explained why it is causing noticeable maintenance problems. In fact, we were informed that the team was struggling with this part both in pre-TDD and post-TDD and the refactoring activities did not address this issue.

Table 7: Pre-TDD

layer	L0	L1	L2	L3	L4
#impl/#var	0/94	4/26	4/12	1/1	142/180

Table 8: SDSM Metrics

layer	L0	L1	L2	L3	L4	L5
#impl/#var	0/145	6/71	5/30	4/17	2/9	1/12
layer	L6	L7	L8	L9	L10	L11
#impl/#var	3/16	3/12	5/12	0/2	2/3	295/399

2. The design rule itself was reflected by a class hierarchy that was shown in multiple levels of the DRH. The DR4 (visitor pattern), DR 6 (MVP), DR 7(a query model), and DR 8 (result sets) belonged to this category. In particular, the MVP design rule (DR 6), which decreased modularity significantly, was realized by 8 interrelated classes (not even interfaces), and many of their implementations depended on each other. DR 4, DR7 and DR 8 were also implemented as class hierarchies, but these classes were loosely coupled. As a result, these design rules constrained the design space by their own hierarchy, and they did not degrade modularity as much as DR6.

After examining the DRH model, the designers realized that DR1, DR2, and DR3 had highest quality: these design rules were realized by interfaces that sit on the top level of DRH, or classes of which interfaces are at top level and implementations are at the bottom level of the DRH.

Summary.

These analyses explained to the developers of the project why the refactored post-TDD did not reveal significant modularity improvement. There were multiple reasons: some design rules are not well-designed (such as DR 6) and some features were poorly implemented (feature 1). These prob-

lems offset the modularity improvement brought by high-quality design rules (such as DR 3). Second, the ADSM metrics, combined with easy feature removal, enabled us to separately examine the impact of each feature or design rule. Finally, the DRH ADSM allowed us to pinpoint poor design decisions, which complemented the metrics.

5. DISCUSSION

In this section, we discuss the threats to validity and our future work.

The problems we identified for this particular project were mainly caused by deep inheritance hierarchies. The DRH-clustered DSMs are particular good at revealing design problems caused by violating the dependency inversion principle. If the system under consideration has other types of design problems, the DRH view may not be as effective.

Although most of the problems identified by our approach were confirmed by the developers, there are a few cases where the developers were not convinced that the having `_impl` variables at higher level was a problem because they believe these particular implementations will remain stable. Our approach is limited in that we do not take the changeability of each module into consideration.

Although the pre-TDD system presents significant difficulty in term of testing and maintenance, the ADSM metrics shows that it is better modularized than post-TDD. The problem is that neither DSM models can reveal the problems caused by large complex objects that takes on too many responsibilities. One of the main reason that motivated the refactoring is the fact that many objects in pre-TDD were too complex to be tested or understood. In our recent work [20], we proposed the *ImpactScope* metrics to assess how well concerns were separated. However, applying this metric requires mapping features to classes. Applying our feature-based metrics remains a future work.

Our experiments on SDSMs were limited by the reverse engineering tool in use. We cannot claim that by removing columns and rows from SDSMs and measuring the resulting matrix directly will not reveal design problems because we do not have a tool that can calculate these properties from the matrix but only from the compiled binaries. However, we suspect that even if we have such a tool, the prevailing metric suite would give mixed results. That is, it is possible that one metric shows improvement but other shows degradation, as we observed from pre-TDD and post-TDD SDSM measurements.

The approach presented in this paper aims to isolate and assess the impact of each individual design rule and feature, which locates the classes where quick and dirty implementation may occur, but not to identify concrete technical debts. Many SDSM-based tools can identify such problems as cyclical dependencies, which is not the focus of this approach.

The ADSM generated from source code may still miss im-

portant decisions that cannot be reflected in classes at all. Our evaluation only shows the benefits of revealing these indirect and implicit dependencies that are extractible from UML class diagrams, but is limited by the limitation of class modeling itself. One of our future work is to combine class models with other high-level models, such as architectural description languages and requirements to improve our ability of architectural decision analysis.

We did not measure the SDSMs using other prevailing metrics, such as the height of inheritance hierarchy. It is possible that by applying and comparing such metrics, we can reveal similar design problems using SDSMs. It is also possible to generalize our DRH algorithm to cluster a SDSM into a lower triangle form, and reveal similar property. Generalizing our DRH algorithm is a future work that will allow us to better assess our approach.

6. CONCLUSION

In this paper, we contributed the formalization and automatic transformation from a UML class diagram to an augmented constraint network and a design structure matrix. We assessed whether these UML-transformed decisions models could better facilitate architecture decision analysis using a real-world software project. The result show that the ADSM and SDSM models possess similar ability of revealing overall modularity properties and their associated metrics complement each other. However, the transformed model enabled designers to view indirect and implicit dependencies that were causing testing and maintenance difficulties, to isolate the impact of features and design rules and to evaluate their quality separately. From the design rule hierarchy view of the ADSM, the designer can also locate the classes that violates important design principles such as dependency inversion.

7. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grants CCF-0916891 and DUE-0837665.

8. REFERENCES

- [1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A challenging model transformation. In *Proc. 10th MODELS*, pages 436–450, Sept. 2007.
- [2] R. Bahsoon and W. Emmerich. Evaluating architectural stability with real options theory. In *Proc. 20th ICSM*, pages 443–447, Sept. 2004.
- [3] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.
- [4] L. Baresi and M. Pezzè. On formalizing UML with high-level petri nets. In *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 271–300. Springer, 2001.
- [5] Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, Aug. 2006.
- [6] Y. Cai, S. Huynh, and T. Xie. A framework and tools support for testing modularity of software design. In *Proc. 22nd ASE*, pages 441–4, Nov. 2007.
- [7] Y. Cai and K. J. Sullivan. Simon: A tool for logical design space modeling and analysis. In *Proc. 20th ASE*, pages 329–332, Nov. 2005.
- [8] Y. Cai and K. J. Sullivan. Modularity analysis of logical design models. In *Proc. 21st ASE*, pages 91–102, Sept. 2006.
- [9] R. Capilla, F. Nava, and C. Carillo. Effort estimation in capturing architectural knowledge. In *Proc. 23rd ASE*, pages 208–217, Sept. 2008.
- [10] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *TSE*, 20(6):476–493, June 1994.
- [11] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. *Computer Standards and Interfaces*, 19(17):325–334, 1998.
- [12] S. Huynh, Y. Cai, and W. Shen. Automatic transformation of UML models into analytical decision models. Technical Report DU-CS-08-01, Drexel University, Apr. 2008. <https://www.cs.drexel.edu/content/uploads/Research/DU-CS-08-01.pdf>.
- [13] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *Proc. 30th ICSE*, pages 411–420, May 2008.
- [14] A. Jansen, J. Bosch, and P. Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, Apr. 2008.
- [15] M. J. LaMantia, Y. Cai, A. D. MacCormack, and J. Rusnak. Analyzing the evolution of large software systems using design structure matrices and design rule theory. In *Proc. 7th WICSA*, pages 83–92, Feb. 2008.
- [16] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, July 2006.
- [17] Object Management Group. Unified modeling language: Superstructure version 2.1.1. <http://www.omg.org/technology/documents/formal/uml.htm>, Feb. 2007.
- [18] Object Management Group. XML metadata interchange version 2.1. <http://www.omg.org/technology/documents/formal/xmi.htm>, Dec. 2007.
- [19] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th OOPSLA*, pages 167–176, Oct. 2005.
- [20] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant’Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. 8th WICSA*, pages 269–272, Sept. 2009.
- [21] D. Varró, S. Gyapay, and A. Pataricza. Automatic transformation of UML models for system verification. In *Proc. Workshop on Trans in UML*, pages 123–127, Apr. 2001.
- [22] S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical models. In *Proc. 24th ASE*, pages 173–184, Nov. 2009.
- [23] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th ASE*, pages 197–208, Nov. 2009.